

# 1. Tipos de datos básicos

**Tipos de datos primitivos:** Son los tipos básicos de datos que Java maneja directamente en el lenguaje y no son objetos.

1. **Enteros:** Son números sin decimales y se dividen en cuatro tipos según su tamaño:

- **byte:** Ocupa 8 bits y su rango va de -128 a 127.
- **short:** Utiliza 16 bits y su rango es de -32,768 a 32,767.
- **int:** Es el tipo entero más utilizado, ocupa 32 bits y su rango va de  $-2^{31}$  a  $2^{31}-1$ .
- **long:** Con 64 bits, su rango es de  $-2^{63}$  a  $2^{63}-1$ .

2. **Punto flotante:** Son números con decimales. Java maneja dos tipos:

- **float:** Ocupa 32 bits y tiene una precisión de 6-7 dígitos decimales significativos.
- **double:** Usa 64 bits y tiene una precisión de aproximadamente 15 dígitos decimales.

3. **Carácter:** Representa un carácter Unicode y utiliza el tipo:

- **char:** Ocupa 16 bits y puede representar un único carácter.

4. **Booleano:** Representa valores de verdad (**true** o **false**) y utiliza el tipo:

- **boolean:** No se define oficialmente su tamaño, pero representa uno de dos posibles valores: **true** o **false**.

**Tipo String:** Aunque no es un tipo primitivo, es uno de los tipos más utilizados en Java. **String** representa una secuencia de caracteres y es, en realidad, un objeto, no un tipo de dato primitivo. Se define de la siguiente manera:

```
String mensaje = "Hola, mundo!";
```

Los objetos **String** son inmutables, lo que significa que una vez creados, su valor no puede cambiar; en su lugar, cualquier modificación produce un nuevo objeto **String**.

**Clases envoltorio (Wrapper classes):** Cada tipo de dato primitivo tiene una clase envoltorio correspondiente en Java. Estas clases "envuelven" el tipo primitivo en un objeto, lo cual es útil cuando necesitas trabajar con un tipo primitivo como si fuera un objeto, por ejemplo, para utilizarlos en colecciones genéricas que requieren objetos, como **ArrayList**. Aquí tienes los tipos primitivos y sus clases envoltorio correspondientes:

- **byte** -> **Byte**
- **short** -> **Short**
- **int** -> **Integer**
- **long** -> **Long**
- **float** -> **Float**
- **double** -> **Double**
- **char** -> **Character**
- **boolean** -> **Boolean**

Las clases envoltorio ofrecen una variedad de métodos útiles, como la conversión de cadenas a tipos primitivos y viceversa, y constantes como **MIN\_VALUE** y **MAX\_VALUE** que representan los valores mínimo y máximo que pueden tener.

Un ejemplo de uso de una clase envoltorio sería:

```
Integer numero = Integer.valueOf(5); // Creando un objeto Integer a partir de un int
int numPrimitivo = numero.intValue(); // Extrayendo el valor primitivo del objeto Integer
```

Las clases envoltorio también permiten trabajar con valores **null**, lo que no es posible con los tipos primitivos. Por ejemplo, un **Integer** puede ser **null**, pero un **int** siempre tendrá un valor numérico, incluso si es **0**.

## 2. Estructuras de control

las estructuras de control en Java nos permiten dirigir el flujo de ejecución del programa de acuerdo con ciertas condiciones. Se pueden clasificar en tres tipos principales:

### 1. Estructuras de selección:

- **if:** Permite ejecutar un bloque de código si se cumple una condición.

```
if (condicion) {  
    // Código a ejecutar si la condición es verdadera  
}
```

- **if-else:** Añade un camino alternativo si la condición **if** no se cumple.

```
if (condicion) {  
    // Código a ejecutar si la condición es verdadera  
} else {  
    // Código a ejecutar si la condición es falsa  
}
```

- **switch:** Permite seleccionar entre múltiples bloques de código a ejecutar.

```
switch (variable) {  
    case valor1:  
        // Código para el caso valor1  
        break;  
    case valor2:  
        // Código para el caso valor2  
        break;  
    // ...  
    default:  
        // Código por defecto si ningún caso coincide  
}
```

### 2. Estructuras de repetición (bucles):

- **for:** Permite repetir un bloque de código un número determinado de veces.

```
for (inicializacion; condicion; incremento) {  
    // Código a repetir  
}
```

- **while:** Ejecuta un bloque de código mientras se cumpla una condición.

```
while (condicion) {  
    // Código a repetir mientras la condición sea verdadera  
}
```

- **do-while:** Similar al **while**, pero garantiza que el bloque de código se ejecute al menos una vez.

```
do {  
    // Código a repetir  
} while (condicion);
```

### 3. Estructuras de salto incondicional:

- **break:** Termina el bucle más interno en el que se encuentra, ya sea **for**, **while** o **do-while**, o un bloque **switch**.
- **continue:** Salta a la siguiente iteración del bucle más interno.
- **return:** Sale del método actual y, opcionalmente, devuelve un valor si el método no es de tipo **void**.

Estas estructuras son fundamentales para cualquier programa, ya que permiten controlar el flujo de ejecución de manera lógica y dinámica en función de diferentes condiciones y escenarios.

## 3. Clases

Las clases en Java son uno de los fundamentos de la programación orientada a objetos (OOP) y se pueden entender como plantillas o moldes que se utilizan para crear objetos. Cada clase define un tipo de dato que incluye atributos (variables) y métodos (funciones o procedimientos) que operan sobre esos atributos o que están relacionados con el comportamiento de la clase.

### Estructura básica de una clase:

```
public class NombreDeLaClase {  
    // Atributos de la clase  
    private int ejemploNumero;  
    private String ejemploCadena;  
  
    // Constructor(es) de la clase  
    public NombreDeLaClase(int numero, String cadena) {  
        this.ejemploNumero = numero;  
        this.ejemploCadena = cadena;  
    }  
  
    // Métodos de la clase  
    public void mostrarDatos() {  
        System.out.println("Número: " + ejemploNumero + ", Cadena: " + ejemploCadena);  
    }  
}
```

### Componentes clave de una clase:

- **Atributos o Campos:** Características o propiedades de la clase, representan el estado de los objetos.
- **Métodos:** Comportamientos o acciones que pueden realizar los objetos de la clase.
- **Constructor(es):** Método especial que se llama para crear una nueva instancia de la clase (un objeto). Puede haber más de un constructor, con diferentes listas de parámetros (sobrecarga de constructores).
- **Bloques de inicialización:** Bloques de código que se ejecutan cuando se crea una instancia de la clase. Son menos comunes y se dividen en bloques de inicialización estática y no estática.

Además, las clases pueden contener:

- **Bloques estáticos:** Se utilizan para inicializar variables estáticas.
- **Clases internas:** Clases definidas dentro de otra clase.

En Java, todo es parte de una clase, y cada aplicación debe tener al menos una clase con un método `main` que actúe como punto de entrada del programa.

```
public class Aplicacion {  
    public static void main(String[] args) {  
        NombreDeLaClase objeto = new NombreDeLaClase(10, "Hola");  
        objeto.mostrarDatos();  
    }  
}
```

En este ejemplo, `Aplicacion` es la clase que contiene el método `main`, y `NombreDeLaClase` es una clase que hemos definido previamente, de la cual creamos un objeto y llamamos a uno de sus métodos.

## 3.1. Atributos o Campos

Los campos en Java, también conocidos como variables de instancia o atributos, son variables que pertenecen a una clase. Estos campos definen las propiedades o el estado que caracterizarán a cada objeto creado a partir de esa clase. Son fundamentales en la programación orientada a objetos, ya que encapsulan los datos que los objetos necesitan para operar.

### Características de los campos en Java:

- **Visibilidad:** Un campo puede tener diferentes niveles de acceso (`public`, `private`, `protected`, o paquete por defecto) que determinan desde dónde puede ser accedido.
- **Estáticos (`static`):** Un campo puede ser marcado como `static`, lo que significa que es compartido por todas las instancias de la clase. Es decir, un campo `static` pertenece a la clase misma, más que a una instancia de la clase.
- **Finales (`final`):** Un campo puede ser declarado como `final`, lo que significa que una vez que se le asigna un valor, no puede ser modificado (es una constante).

### Ejemplo de declaración de campos en una clase:

```
public class Estudiante {
    // Campo privado, solo accesible dentro de la clase Estudiante
    private String nombre;

    // Campo estático, compartido por todas las instancias de la clase
    private static int contadorEstudiantes = 0;

    // Constructor que actualiza el nombre y el contador de estudiantes
    public Estudiante(String nombre) {
        this.nombre = nombre;
        contadorEstudiantes++;
    }

    // Métodos para acceder y modificar el nombre (getters y setters)
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    // Método estático para obtener el número total de estudiantes
    public static int getContadorEstudiantes() {
        return contadorEstudiantes;
    }
}
```

En este ejemplo, cada objeto `Estudiante` tiene su propio `nombre`, pero comparten un único `contadorEstudiantes` porque es un campo `static`. Además, el campo `nombre` es `private`, lo que significa que solo puede ser accedido directamente dentro de la clase `Estudiante`, y se proporcionan métodos `public` para obtener y modificar su valor desde fuera de la clase.

## 3.2. Métodos

Los métodos en Java son bloques de código que tienen un nombre y pueden ser ejecutados al ser llamados. Son similares a las funciones en otros lenguajes de programación y sirven para realizar tareas específicas, permitiendo así la reutilización de código y una mejor organización del programa.

Aquí tienes los componentes principales de un método en Java:

- **Nombre del método:** Identifica el método y se utiliza para llamarlo.
- **Parámetros (argumentos):** Son las entradas que el método puede aceptar para realizar su tarea. Los parámetros son opcionales; un método puede no tener ninguno.
- **Tipo de retorno:** Indica el tipo de dato que el método devolverá como resultado de su ejecución. Si un método no devuelve ningún valor, su tipo de retorno es `void`.
- **Cuerpo del método:** Es el bloque de código que contiene las instrucciones que se ejecutan cuando el método es llamado.

Aquí tienes un ejemplo simple de un método en Java:

```
public int sumar(int numero1, int numero2) {  
    int resultado = numero1 + numero2;  
    return resultado; // Devuelve el resultado de la suma  
}
```

En este ejemplo, `sumar` es el nombre del método, `int numero1` y `int numero2` son los parámetros, `int` es el tipo de retorno, y el cuerpo del método es todo lo que está entre llaves `{}`.

Los métodos pueden ser llamados desde otras partes del código, y si están marcados como `static`, pueden ser llamados sin necesidad de crear una instancia de la clase. Si no son `static`, se deben llamar a través de un objeto de la clase donde el método está definido.

```
public class Calculadora {  
    public static void main(String[] args) {  
        Calculadora calc = new Calculadora();  
        int suma = calc.sumar(5, 10);  
        System.out.println("La suma es: " + suma);  
    }  
}
```

En este código, se crea una instancia de `Calculadora` y se llama al método `sumar` sobre esa instancia, luego se imprime el resultado. Los métodos pueden ser `public`, `private`, `protected`, o tener el acceso por defecto (sin modificador), lo que determina dónde y cómo pueden ser accedidos dentro del programa.

## Sobrecarga de métodos

La sobrecarga de métodos, o "method overloading", es una característica de Java que permite definir varios métodos con el mismo nombre en una clase, siempre y cuando cada método tenga una lista de parámetros diferente. La sobrecarga facilita la legibilidad del código y mejora la experiencia del programador al poder usar el mismo nombre de método para realizar tareas similares pero con diferentes tipos o cantidades de argumentos.

**Puntos clave de la sobrecarga de métodos:**

- **Diferentes firmas:** Para sobrecargar un método, se deben variar los tipos, el número o el orden de los parámetros.
- **Mismo nombre de método:** Los métodos sobrecargados comparten el mismo nombre dentro de la misma clase.
- **Tipo de retorno no relevante:** El tipo de retorno no se considera para la sobrecarga de métodos, es decir, dos métodos no pueden ser sobrecargados solo cambiando su tipo de retorno.

**Ejemplo de sobrecarga de métodos en Java:**

```
public class Calculadora {  
  
    // Método para sumar dos enteros  
    public int sumar(int a, int b) {  
        return a + b;  
    }  
  
    // Método sobrecargado para sumar tres enteros  
    public int sumar(int a, int b, int c) {  
        return a + b + c;  
    }  
  
    // Método sobrecargado para sumar dos números reales (double)  
    public double sumar(double a, double b) {  
        return a + b;  
    }  
}
```

En este ejemplo, la clase `Calculadora` tiene tres versiones del método `sumar`, cada una con diferentes parámetros. Java determina cuál método utilizar en tiempo de compilación, basándose en la correspondencia entre los argumentos utilizados en la llamada del método y los parámetros definidos en las firmas de los métodos sobrecargados.

## 3.3. Constructores e instanciación

En Java, un constructor es un bloque de código especial dentro de una clase que se llama automáticamente cuando se crea una nueva instancia de esa clase. Los constructores tienen el mismo nombre que la clase y no tienen tipo de retorno, ni siquiera `void`.

### Características de los constructores:

- **Inicialización:** Los constructores se utilizan para inicializar el estado de un objeto, es decir, asignar valores a los campos o realizar cualquier configuración inicial necesaria.
- **Sobrecarga:** Al igual que los métodos, los constructores también pueden ser sobrecargados, permitiendo diferentes formas de crear instancias de una clase con distintos estados iniciales.
- **Constructor por defecto:** Si no se define un constructor explícitamente en una clase, Java proporciona un constructor por defecto sin argumentos que simplemente crea una instancia de la clase.

**Instanciación** es el proceso de crear un nuevo objeto o instancia de una clase utilizando el operador `new` seguido por la llamada a uno de los constructores de la clase.

### Ejemplo de constructor e instanciación en Java:

```
public class Persona {
    private String nombre;
    private int edad;

    // Constructor de la clase Persona
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Métodos getters y setters...
}

// Creando una instancia de la clase Persona
public class Principal {
    public static void main(String[] args) {
        Persona persona = new Persona("Ana", 30); // Instanciación y llamada al constructor
        // Ahora 'persona' es una instancia de la clase Persona con nombre "Ana" y edad 30.
    }
}
```

En este ejemplo, `Persona(String nombre, int edad)` es el constructor de la clase `Persona` que inicializa los campos `nombre` y `edad`. En la clase `Principal`, se crea una nueva instancia de `Persona` llamando a su constructor con los argumentos "Ana" y 30.

### 3.4. Resumen de Modificadores

Modificador	Clase	Paquete	Subclase	Mundo	Descripción
public	Sí	Sí	Sí	Sí	Accesible desde cualquier otra clase.
protected	Sí	Sí	Sí	No	Accesible dentro del mismo paquete y subclases en otros paquetes.
(default)	Sí	Sí	No	No	Sin modificador; accesible solo dentro del mismo paquete.
private	Sí	No	No	No	Accesible solo dentro de la misma clase.
static	Sí	Sí	Sí	Sí	Indica que el miembro pertenece a la clase, en lugar de una instancia de la clase.
final	Sí	Sí	Sí	Sí	Usado para definir una constante, un método que no puede ser sobrescrito, o una clase que no puede ser subclassificada.
abstract	Sí	Sí	Sí	Sí	Usado para declarar una clase que no puede ser instanciada o un método que debe ser implementado por clases hijas.
transient	No	Sí	Sí	Sí	Evita que un campo sea serializado.
volatile	No	Sí	Sí	Sí	Indica que una variable puede ser cambiada inesperadamente, especialmente en contextos de múltiples hilos.

- **Modificador:** El tipo de modificador aplicado a clases, variables o métodos.
- **Clase:** Indica si el modificador puede ser aplicado a una clase.
- **Paquete:** Accesibilidad dentro del mismo paquete.
- **Subclase:** Accesibilidad desde subclases, incluso fuera del paquete.
- **Mundo:** Accesibilidad desde cualquier parte fuera del paquete.
- **Descripción:** Breve descripción de lo que el modificador implica o cómo afecta la accesibilidad y el comportamiento del componente al que se aplica.

## 4. Encapsulación y visibilidad

La encapsulación es uno de los cuatro principios fundamentales de la programación orientada a objetos y se refiere a la práctica de ocultar los detalles de la implementación de una clase y exponer solo una interfaz pública. Esto se logra utilizando diferentes niveles de visibilidad para los campos y métodos de una clase.

**Visibilidad:** En Java, hay cuatro niveles de visibilidad que se pueden asignar a campos y métodos:

- **public:** El campo o método es accesible desde cualquier otra clase.
- **protected:** El acceso está limitado a las clases dentro del mismo paquete y a todas las subclases.
- **default** (sin modificador): El acceso está limitado a las clases dentro del mismo paquete.
- **private:** El campo o método solo es accesible dentro de la misma clase.

**Encapsulación:** Al hacer privados los campos (variables de instancia) de una clase, se restringe el acceso directo a ellos desde fuera de la clase, lo que evita que el estado interno del objeto pueda ser alterado de manera inesperada o incorrecta. En su lugar, se proporcionan métodos públicos (generalmente getters y setters) para obtener y modificar esos valores de forma controlada.

**Ejemplo de encapsulación en Java:**

```
public class CuentaBancaria {
    // Campo privado: detalle de implementación oculto
    private double saldo;

    // Constructor público: permite inicializar objetos
    public CuentaBancaria(double saldoInicial) {
        this.saldo = saldoInicial;
    }

    // Método público: permite consultar el saldo (getter)
    public double getSaldo() {
        return saldo;
    }

    // Método público: permite depositar dinero (setter)
    public void depositar(double cantidad) {
        if (cantidad > 0) {
            saldo += cantidad;
        }
    }

    // Otros métodos...
}
```

En este ejemplo, el campo `saldo` es privado, lo que significa que no se puede acceder ni modificar directamente desde fuera de la clase `CuentaBancaria`. En su lugar, se utilizan métodos públicos para interactuar con ese campo, proporcionando una interfaz controlada para operaciones como consultar el saldo o depositar dinero. La encapsulación ayuda a proteger la integridad de los datos y facilita el mantenimiento y la evolución del software.

## 5. Herencia

La herencia es un principio de la programación orientada a objetos donde una clase (llamada subclase o clase derivada) puede heredar campos y métodos de otra clase (llamada superclase o clase base). La herencia facilita la reutilización de código y la creación de una jerarquía de clases.

En Java, la herencia se realiza utilizando la palabra clave `extends`. La subclase hereda todos los miembros accesibles (campos y métodos) de la superclase, excepto los constructores. Los constructores no se heredan porque están vinculados al nombre de la clase, y la subclase tiene un nombre diferente.

### Constructores y herencia:

Cuando se crea una instancia de una subclase, Java asegura que la parte de la superclase de esa instancia también se inicialice correctamente. Esto se logra llamando a uno de los constructores de la superclase, ya sea explícita o implícitamente.

- **Constructor por defecto:** Si no se especifica un constructor en la subclase, se llama implícitamente al constructor por defecto de la superclase.
- **Llamada a `super()`:** Si la superclase no tiene un constructor por defecto o si se necesita llamar a un constructor específico, se debe usar `super()` con los argumentos adecuados en la primera línea del constructor de la subclase.

### Ejemplo de herencia y constructores en Java:

```
// Superclase
public class Vehiculo {
    private String marca;

    // Constructor de la superclase
    public Vehiculo(String marca) {
        this.marca = marca;
    }

    // Métodos de la superclase...
}

// Subclase
public class Coche extends Vehiculo {
    private int numeroDePuertas;

    // Constructor de la subclase
    public Coche(String marca, int numeroDePuertas) {
        super(marca); // Llamada al constructor de la superclase
        this.numeroDePuertas = numeroDePuertas;
    }

    // Métodos de la subclase...
}
```

En este ejemplo, `Coche` es una subclase de `Vehiculo`. El constructor de `Coche` llama al constructor de `Vehiculo` usando `super(marca)` para asegurarse de que la parte `Vehiculo` del objeto `Coche` se inicialice correctamente. La subclase puede entonces añadir su propia inicialización específica, en este caso, estableciendo el número de puertas.

## 5.1. Acceso a la superclase y sobreescritura

En Java, cuando trabajamos con herencia, a menudo necesitamos acceder a propiedades y métodos de la superclase desde una subclase. Esto se puede hacer directamente si la visibilidad de esos miembros lo permite (es decir, si son `public` o `protected`). Además, las subclases pueden sobrescribir métodos heredados para proporcionar una implementación específica que se ajuste a su comportamiento particular.

### Acceso a propiedades y métodos de la superclase:

Para acceder a los miembros no privados de la superclase, simplemente los usamos como si fueran parte de la subclase. Si queremos referirnos explícitamente a un miembro de la superclase, podemos usar la palabra clave `super`. Esto es especialmente útil cuando un campo o método se oculta o se sobrescribe en la subclase.

```
>
public class Superclase {
    protected String propiedad;

    public void mostrar() {
        System.out.println("Propiedad en Superclase: " + propiedad);
    }
}

public class Subclase extends Superclase {
    public void usarPropiedadSuperclase() {
        super.propiedad = "Valor desde Subclase";
        super.mostrar(); // Llama al método de la superclase
    }
}
```

### Sobrescritura de métodos (Overriding):

La sobrescritura de métodos ocurre cuando una subclase proporciona una implementación específica para un método que ya está definido en su superclase. El método sobrescrito en la subclase debe tener la misma firma que el método en la superclase (mismo nombre, misma lista de parámetros y, a partir de Java 5, también el mismo tipo de retorno si este es covariante).

Para indicar y asegurarse de que un método está siendo sobrescrito correctamente, se puede anotar con `@Override`.

```
public class Superclase {
    public void metodo() {
        System.out.println("Método en Superclase.");
    }
}

public class Subclase extends Superclase {
    @Override
    public void metodo() {
        System.out.println("Método sobrescrito en Subclase.");
    }
}
```

En este ejemplo, `Subclase` sobrescribe el método `metodo()`. Cuando se llame a `metodo()` en un objeto de tipo `Subclase`, se ejecutará la versión sobrescrita, no la versión original de `Superclase`. Esto permite que las subclases personalicen o mejoren el comportamiento de los métodos heredados.

## 6. Clases y métodos abstracto

En Java, una clase abstracta es una clase que no se puede instanciar por sí misma y está destinada a ser una superclase de otras clases. Las clases abstractas se utilizan como base para otras clases, permitiéndoles compartir una estructura común y comportamiento, definiendo al mismo tiempo que ciertos métodos deben ser implementados por las subclases.

### Clases Abstractas:

- Para declarar una clase como abstracta, se utiliza la palabra clave `abstract`.
- Puede contener tanto métodos abstractos como métodos no abstractos.
- Un método abstracto es un método que no tiene implementación en la clase abstracta; sólo se proporciona su firma.
- Si una clase contiene al menos un método abstracto, entonces la clase debe ser declarada como abstracta.
- No se pueden crear objetos directamente de una clase abstracta; es necesario extenderla y proporcionar implementaciones concretas de los métodos abstractos en una subclase.

### Métodos Abstractos:

- Un método abstracto se declara sin un cuerpo y termina con un punto y coma (;) en lugar de un bloque de código.
- Los métodos abstractos actúan como un contrato, obligando a las subclases a implementar estos métodos con un comportamiento específico.

### Ejemplo de clase y método abstracto en Java:

```
// Clase abstracta
public abstract class Figura {
    // Método abstracto
    public abstract double calcularArea();

    // Método concreto
    public void imprimir() {
        System.out.println("Figura con área: " + calcularArea());
    }
}

// Subclase concreta
public class Circulo extends Figura {
    private double radio;

    public Circulo(double radio) {
        this.radio = radio;
    }

    // Implementación del método abstracto
    @Override
    public double calcularArea() {
        return Math.PI * radio * radio;
    }
}

public class Main {
    public static void main(String[] args) {
        // No se puede instanciar Figura: Figura f = new Figura();
        Circulo c = new Circulo(5);
        c.imprimir(); // "Figura con área: 78.53981633974483"
    }
}
```

En este ejemplo, `Figura` es una clase abstracta que define un método abstracto `calcularArea()`. La clase `Circulo` extiende `Figura` y proporciona una implementación concreta para `calcularArea()`. No se puede crear una instancia de `Figura` directamente, pero sí de `Circulo`, que implementa todos los métodos abstractos heredados de `Figura`.

## 7. Clases y métodos finales

En Java, la palabra clave `final` tiene varios usos, pero cuando se refiere a clases y métodos, se usa para restringir la herencia y la sobrescritura, respectivamente.

### Clases Finales:

Una clase declarada como final no puede ser extendida. En otras palabras, no puedes crear una subclase de una clase final. El uso de una clase final es una forma de asegurar que la implementación de esa clase permanezca inalterada y constante, ya que nadie puede cambiar su comportamiento mediante herencia.

```
public final class MiClaseFinal {  
    // Detalles de la clase...  
}
```

Por ejemplo, `String` es una clase final en Java, lo que significa que no puedes extenderla para crear una subclase de `String`.

### Métodos Finales:

Un método final no puede ser sobrescrito por las subclases. Esto es útil cuando quieres asegurarte de que el comportamiento específico de un método permanezca igual para todas las subclases, conservando así la intención original del diseño de la clase.

```
public class MiClase {  
    public final void metodoFinal() {  
        // Detalles del método...  
    }  
}
```

En este caso, cualquier clase que extienda `MiClase` no podrá sobrescribir `metodoFinal()`.

Es importante mencionar que si bien puedes tener una clase final con métodos no finales, no tiene sentido tener una clase abstracta final, ya que no se podría extender para implementar los métodos abstractos, y tampoco tendría sentido tener un método abstracto final, ya que esto se contradiría, porque un método abstracto necesita ser sobrescrito para ser utilizado.

## 8. Interfaces

En Java, una interfaz es un tipo de referencia que es similar a una clase abstracta en el sentido de que puede contener métodos abstractos que las clases pueden implementar. No obstante, a diferencia de las clases abstractas, las interfaces no pueden tener estado; es decir, no pueden contener atributos que no sean constantes (todos los campos en una interfaz son por defecto públicos, estáticos y finales).

Las interfaces son muy útiles para definir un contrato que las clases deben cumplir, especificando qué métodos deben ser implementados, sin imponer cómo deben ser implementados. A partir de Java 8, las interfaces también pueden contener métodos predeterminados (default) con implementación y métodos estáticos con cuerpo.

### Características clave de las interfaces en Java:

- Todos los métodos de una interfaz son por defecto abstractos y públicos, a menos que se declare explícitamente como default o static.
- Una clase puede implementar múltiples interfaces, lo que permite una forma de múltiple herencia de tipos.
- Al implementar una interfaz, una clase debe proporcionar implementaciones concretas para todos los métodos abstractos de la interfaz.
- Las interfaces no pueden ser instanciadas; sin embargo, se pueden utilizar como tipos de referencia para objetos de cualquier clase que las implemente.

### Ejemplo de una interfaz en Java:

```
public interface Vehiculo {

    // Método abstracto
    void acelerar(int incremento);

    // Método por defecto (Java 8+)
    default void frenar(int decremento) {
        System.out.println("Frenando el vehículo por " + decremento + " unidades.");
    }
}

public class Coche implements Vehiculo {
    private int velocidad;

    // Constructor
    public Coche() {
        this.velocidad = 0;
    }

    // Implementación del método abstracto de la interfaz
    @Override
    public void acelerar(int incremento) {
        velocidad += incremento;
        System.out.println("Acelerando a " + velocidad + " km/h.");
    }
}

public class Main {
    public static void main(String[] args) {
        Coche miCoche = new Coche();
        miCoche.acelerar(30); // "Acelerando a 30 km/h."
        miCoche.frenar(10);   // "Frenando el vehículo por 10 unidades."
    }
}
```

En este ejemplo, la interfaz **Vehiculo** define un método abstracto **acelerar** y un método por defecto **frenar**. La clase **Coche** implementa la interfaz **Vehiculo**, proporcionando su propia implementación del método **acelerar** y heredando el comportamiento del método **frenar**.

## 9. Polimorfismo

El polimorfismo es un concepto fundamental en la programación orientada a objetos y en Java, donde "poli" significa muchos y "morfismo" significa formas. Se refiere a la capacidad de una variable, función o objeto de tomar múltiples formas. En Java, el polimorfismo se manifiesta principalmente de dos maneras: polimorfismo de métodos y polimorfismo de herencia.

**Polimorfismo de Métodos:** Incluye la sobrecarga de métodos (overloading) y la sobrescritura de métodos (overriding).

- **Sobrecarga de Métodos (Overloading):** Ocurre cuando varios métodos en una clase tienen el mismo nombre pero diferentes listas de parámetros (diferentes tipos de parámetros o número de parámetros). La sobrecarga permite que un método realice diferentes tipos de tareas en función de los parámetros con los que se invoca.
- **Sobrescritura de Métodos (Overriding):** Sucede cuando una subclase proporciona una implementación específica de un método que ya está definido en su superclase o en una interfaz que implementa.

**Polimorfismo de Herencia:** Se refiere a la capacidad de una clase para proporcionar diferentes implementaciones de métodos que son llamados a través de referencias de tipo de superclase. Esto significa que se puede usar una referencia de tipo de superclase para referirse a un objeto de una subclase.

**Ejemplo de Polimorfismo en Java:**

```
class Animal {
    public void sonido() {
        System.out.println("El animal hace un sonido");
    }
}

class Perro extends Animal {
    @Override
    public void sonido() {
        System.out.println("El perro ladra");
    }
}

class Gato extends Animal {
    @Override
    public void sonido() {
        System.out.println("El gato maúlla");
    }
}

public class TestPolimorfismo {
    public static void main(String[] args) {
        Animal miAnimal = new Animal();
        Animal miPerro = new Perro();
        Animal miGato = new Gato();

        miAnimal.sonido(); // El animal hace un sonido
        miPerro.sonido();  // El perro ladra
        miGato.sonido();   // El gato maúlla
    }
}
```

En este ejemplo, se utiliza el polimorfismo para referirse a objetos de las clases **Perro** y **Gato** a través de referencias de tipo **Animal**. Cuando se invoca el método **sonido()**, se ejecuta la versión del método correspondiente al tipo real del objeto al que apunta la referencia, lo cual es un ejemplo de ligadura dinámica o de tiempo de ejecución.

## 10. Arrays

Un array en Java es una estructura de datos que te permite almacenar múltiples valores del mismo tipo en una sola variable. Es una colección de variables de tipo fijo que están indexadas y cuyo tamaño se establece cuando se crea el array y no se puede cambiar después. Cada elemento en un array es accesible a través de su índice, el cual comienza desde 0.

### Características de los arrays en Java:

- Tienen un tamaño fijo que se define en el momento de su creación.
- Pueden almacenar datos primitivos (como `int`, `double`, `char`, etc.) o referencias a objetos.
- Son objetos en Java, por lo que se crean con el operador `new`.
- Permiten el acceso rápido a los elementos utilizando el índice.

### Ejemplo de declaración, creación e inicialización de un array en Java:

```
// Declaración de un array de enteros
int[] miArray;

// Creación del array de enteros con un tamaño de 5
miArray = new int[5];

// Inicialización de los valores del array
miArray[0] = 10;
miArray[1] = 20;
miArray[2] = 30;
miArray[3] = 40;
miArray[4] = 50;

// Acceso a los elementos del array
System.out.println(miArray[2]); // Salida: 30

// También se puede declarar, crear e inicializar en una sola línea
int[] otroArray = {10, 20, 30, 40, 50};

// Recorrer un array con un bucle for
for(int i = 0; i < miArray.length; i++) {
    System.out.println(miArray[i]);
}
```

Recuerda que al intentar acceder a un índice fuera de los límites del array (por ejemplo, `miArray[5]` en un array de tamaño 5), se lanzará una excepción `ArrayIndexOutOfBoundsException`.

## 10.1. Resumen de métodos

La clase `Arrays` en Java, que se encuentra en el paquete `java.util`, proporciona una serie de métodos estáticos para realizar operaciones comunes en arrays, como ordenarlos, buscar en ellos y rellenarlos, entre otros. Al ser métodos estáticos, se pueden invocar directamente desde la clase `Arrays` sin necesidad de crear una instancia de ella.

Aquí tienes un resumen de algunos de los métodos más comunes de la clase `Arrays`:

- **`sort()`**: Ordena un array completo o un rango especificado dentro de un array. Se puede usar para tipos primitivos o para objetos que implementen la interfaz `Comparable` o que tengan un `Comparator`.
- **`binarySearch()`**: Realiza una búsqueda binaria en un array ordenado y devuelve el índice del elemento buscado si se encuentra. Si no se encuentra, devuelve un índice negativo.
- **`equals()`**: Compara dos arrays para determinar si son iguales, es decir, si tienen la misma longitud y todos sus elementos correspondientes son iguales.
- **`fill()`**: Asigna un valor específico a cada elemento de un array completo o a un rango de un array.
- **`copyOf()`**: Crea una nueva copia de un array, con la longitud que se especifique.
- **`copyOfRange()`**: Crea una copia parcial de un array, desde el índice inicial hasta el índice final especificados.
- **`toString()`**: Devuelve una representación en forma de cadena de texto de un array, útil para imprimir el contenido del array.
- **`asList()`**: Convierte un array en una lista (`List`).
- **`deepEquals()`**: Similar a `equals()`, pero diseñado para [arrays multidimensionales](#), donde compara recursivamente cada subarray.
- **`hashCode()`**: Calcula el valor de `hashCode` para un array, basado en el contenido de todos sus elementos.
- **`deepHashCode()`**: Similar a `hashCode()`, pero para [arrays multidimensionales](#).
- **`stream()`**: A partir de Java 8, se puede utilizar para crear un flujo (`Stream`) a partir de un array, lo que permite trabajar con APIs de flujo.

Estos métodos son muy útiles para trabajar con arrays y te permiten realizar tareas comunes de manera eficiente y con menos código. Aquí te pongo un ejemplo de cómo podrías utilizar algunos de estos métodos:

```
import java.util.Arrays;

public class EjemploArrays {
    public static void main(String[] args) {
        // Crear y rellenar un array
        int[] numeros = new int[10];
        Arrays.fill(numeros, 5);

        // Ordenar el array
        Arrays.sort(numeros);

        // Buscar un valor en el array ordenado
        int indice = Arrays.binarySearch(numeros, 5);

        // Comparar dos arrays para ver si son iguales
        int[] otroArray = {5, 5, 5, 5, 5, 5, 5, 5, 5, 5};
        boolean sonIguales = Arrays.equals(numeros, otroArray);

        // Convertir un array a una cadena de texto
        String arrayComoTexto = Arrays.toString(numeros);

        // Imprimir resultados
        System.out.println("Índice del elemento 5: " + indice);
        System.out.println("¿Son iguales los arrays? " + sonIguales);
        System.out.println("Array como texto: " + arrayComoTexto);
    }
}
```

Recuerda que para usar la clase `Arrays` necesitas importarla al principio de tu archivo de código fuente con la línea `import java.util.Arrays;`.

# 11. Strings

Las cadenas de caracteres en Java, representadas por la clase `String`, son secuencias inmutables de caracteres. La inmutabilidad significa que una vez que se crea una cadena de caracteres, no se puede cambiar su contenido. Java gestiona las cadenas de forma eficiente mediante el uso del "pool de strings", que es una ubicación especial de la memoria donde almacena las instancias de cadenas literales.

## Características de las cadenas de caracteres (Strings) en Java:

- **Inmutabilidad:** Cada vez que se realiza una operación que modifica una cadena de caracteres, como una concatenación o reemplazo, en realidad se está creando una nueva cadena en lugar de cambiar la existente.
- **Pool de Strings:** Java mantiene un conjunto de cadenas únicas que han sido utilizadas en el programa para optimizar la memoria. Si se crea una cadena de caracteres que ya existe en el pool, Java reutiliza la instancia en lugar de crear una nueva.
- **Operaciones comunes:** La clase `String` ofrece una amplia gama de métodos para realizar operaciones comunes como comparar cadenas (`equals`, `compareTo`), buscar subcadenas (`indexOf`, `lastIndexOf`), extraer subcadenas (`substring`), convertir a mayúsculas o minúsculas (`toUpperCase`, `toLowerCase`), entre otros.
- **Concatenación:** Se pueden unir dos cadenas usando el operador `+`. Internamente, Java usa `StringBuilder` o `StringBuffer` para realizar la concatenación de manera eficiente.
- **Conversión:** Las cadenas pueden convertirse desde y hacia otros tipos de datos, como primitivos (`parseInt`, `toString`) y arrays de caracteres (`toCharArray`).

## Ejemplo de creación y uso de cadenas de caracteres en Java:

```
String saludo = "Hola, ";
String nombre = "Mundo";
String mensaje = saludo + nombre; // Concatenación
System.out.println(mensaje); // Imprime "Hola, Mundo"

boolean sonIguales = "test".equals("test"); // Comparación de cadenas
System.out.println(sonIguales); // Imprime true

String enMayusculas = mensaje.toUpperCase(); // Convertir a mayúsculas
System.out.println(enMayusculas); // Imprime "HOLA, MUNDO"

int longitud = nombre.length(); // Longitud de la cadena
System.out.println("Longitud de '" + nombre + "': " + longitud); // Imprime 5

char letra = nombre.charAt(0); // Obtener el carácter en el índice 0
System.out.println("Primera letra de '" + nombre + "': " + letra); // Imprime 'M'
```

En resumen, las cadenas de caracteres son una parte fundamental de la mayoría de las aplicaciones Java, y es importante comprender su inmutabilidad y cómo manipular su contenido utilizando los métodos proporcionados por la clase `String`.

## 11.1. Resumen de métodos

La clase `String` en Java tiene una variedad de métodos que te permiten manipular cadenas de caracteres. Aquí tienes un resumen con algunos de los métodos más utilizados:

- **`length()`**: Retorna la longitud de la cadena de caracteres.
- **`charAt(int index)`**: Retorna el carácter en la posición especificada.
- **`substring(int beginIndex, int endIndex)`**: Retorna una nueva cadena que es una subcadena de esta cadena.
- **`concat(String str)`**: Concatena la cadena especificada al final de esta cadena.
- **`indexOf(int ch)` y `lastIndexOf(int ch)`**: Retorna el índice de la primera/última aparición del carácter especificado o -1 si no se encuentra.
- **`indexOf(String str)` y `lastIndexOf(String str)`**: Similar a los anteriores, pero buscando una subcadena.
- **`startsWith(String prefix)` y `endsWith(String suffix)`**: Comprueba si la cadena comienza o termina con la subcadena especificada.
- **`toLowerCase()` y `toUpperCase()`**: Convierte todos los caracteres de la cadena a minúsculas o mayúsculas, respectivamente.
- **`equals(Object anObject)`**: Compara esta cadena con el objeto especificado para la igualdad.
- **`equalsIgnoreCase(String anotherString)`**: Compara esta cadena con otra cadena, ignorando diferencias de mayúsculas y minúsculas.
- **`contains(CharSequence s)`**: Retorna true si la cadena contiene la secuencia de caracteres especificada.
- **`replace(char oldChar, char newChar)`**: Retorna una nueva cadena resultante de reemplazar todas las ocurrencias del carácter antiguo en esta cadena con el nuevo carácter.
- **`replaceAll(String regex, String replacement)`**: Reemplaza cada subcadena de esta cadena que coincida con la expresión regular dada con la cadena de reemplazo dada.
- **`split(String regex)`**: Divide esta cadena alrededor de coincidencias de la expresión regular dada.
- **`trim()`**: Retorna una copia de la cadena con espacios en blanco iniciales y finales eliminados.

### Ejemplo de uso de algunos métodos de la clase `String`:

```
String saludo = "Hola Mundo";

// Longitud de la cadena
int longitud = saludo.length();

// Carácter en una posición específica
char caracter = saludo.charAt(5);

// Subcadena
String mundo = saludo.substring(5);

// Concatenación
String exclamacion = saludo.concat("!");

// Búsqueda de subcadena
int indiceDeMundo = saludo.indexOf("Mundo");

// Comprobación de igualdad
boolean esIgual = saludo.equals("hola mundo");

// Ignorar mayúsculas/minúsculas en la comparación
boolean esIgualIgnorandoMayusculas = saludo.equalsIgnoreCase("hola mundo");

// Reemplazo de caracteres
String saludoFestivo = saludo.replace("Mundo", "Festivo");

// División en subcadenas
String[] palabras = saludo.split(" ");

// Eliminación de espacios
String saludoSinEspacios = saludo.trim();
```

Es importante recordar que los métodos de la clase `String` no modifican la cadena original, ya que las cadenas en Java son inmutables. En lugar de eso, estos métodos devuelven una nueva cadena que es el resultado de la operación realizada.

## 12. Colecciones

Las colecciones en Java son estructuras de datos que sirven para almacenar grupos de objetos. En lugar de utilizar arrays fijos, las colecciones ofrecen una manera más flexible de trabajar con conjuntos de datos. La [biblioteca](#) de colecciones de Java proporciona diferentes tipos de colecciones, cada una con sus características y usos específicos.

La API de Colecciones de Java se encuentra principalmente en el paquete `java.util` e incluye interfaces, implementaciones y algoritmos que pueden trabajar con ellas. Las interfaces principales son:

- **List**: Una colección ordenada que puede contener elementos duplicados. Permite el acceso posicional a los elementos. Ejemplos de implementaciones son `ArrayList` y `LinkedList`.
- **Set**: Una colección que no permite elementos duplicados. No garantiza orden en la iteración. Ejemplos son `HashSet` y `TreeSet`.
- **Queue**: Una colección diseñada para mantener elementos antes de su procesamiento. Implementa operaciones de tipo FIFO (primero en entrar, primero en salir). Un ejemplo es `LinkedList`, que implementa la interfaz `Queue`.
- **Map**: No es técnicamente una colección, pero es parte de la [biblioteca](#) de colecciones. Un `Map` almacena pares clave-valor y no permite claves duplicadas. Ejemplos son `HashMap` y `TreeMap`.

Además de estas interfaces, la [biblioteca](#) de colecciones proporciona clases de utilidad como `Collections`, que ofrece métodos estáticos para operar en colecciones, como ordenar y buscar, y `Arrays`, que proporciona métodos estáticos para operar en arrays.

Las colecciones también admiten iteradores, que son objetos que permiten recorrer los elementos de una colección de manera secuencial.

## 12.1. List

Las colecciones de tipo **List** en Java son estructuras de datos que forman parte del framework de colecciones y representan una secuencia ordenada de elementos. Pueden contener elementos duplicados, y cada elemento tiene un índice basado en su posición que comienza desde cero, lo que permite un acceso posicional preciso a los elementos.

Las principales características de las **List** son:

- **Acceso posicional:** Puedes obtener, establecer o añadir elementos en posiciones específicas de la lista.
- **Búsqueda:** Puedes buscar elementos y obtener su índice dentro de la lista.
- **Iteración:** Puedes recorrer la lista de forma secuencial utilizando un iterador o un bucle **for-each**.
- **Rango de operaciones:** Puedes realizar operaciones en subconjuntos de la lista, como **subList**.
- **Elementos duplicados:** Las listas permiten elementos repetidos y mantienen su orden de inserción.

Aquí tienes una breve descripción de las implementaciones más comunes de la interfaz **List**: **ArrayList**, **Stack**, **Vector** y **LinkedList**.

### 1. **ArrayList**:

- Es una implementación de la interfaz **List** que utiliza un array que crece dinámicamente para almacenar los elementos.
- Permite acceso aleatorio rápido a los elementos con un tiempo constante para la operación **get**, que es ideal para situaciones donde se requiere lectura frecuente.
- No es sincronizado, por lo que no es seguro para hilos sin sincronización externa.
- La adición y eliminación de elementos puede ser costosa si no se realiza al final de la lista, ya que puede requerir desplazamientos internos.

### 2. **Stack**:

- Es una subclase de **Vector** que implementa una estructura de datos tipo pila (LIFO - Último en entrar, primero en salir).
- Proporciona métodos como **push** para agregar elementos y **pop** para eliminar y retornar el elemento superior de la pila.
- Es una clase que se considera algo obsoleta en Java (se recomienda usar **Deque** en su lugar).
- Es sincronizado, lo que implica que es seguro para hilos pero puede ser menos eficiente en entornos sin contienda.

### 3. **Vector**:

- Similar a **ArrayList**, pero con métodos sincronizados para operaciones seguras en entornos multihilo.
- Cada operación es segura para hilos, lo que puede llevar a una disminución del rendimiento en contextos donde la sincronización no es necesaria.
- Al igual que **ArrayList**, permite acceso aleatorio rápido a elementos.

### 4. **LinkedList**:

- Implementa tanto la interfaz **List** como **Deque**, y puede usarse tanto como lista como cola (FIFO - Primero en entrar, primero en salir) o pila (LIFO).
- Utiliza una lista enlazada doblemente para almacenar los elementos, lo que facilita la inserción y eliminación de elementos en cualquier parte de la lista, especialmente al principio y al final.
- El acceso a los elementos es secuencial, por lo que puede ser más lento en comparación con **ArrayList** y **Vector**.
- No es sincronizada, por lo que no es segura para hilos sin manejo externo.

Aquí tienes un pequeño ejemplo de cómo podrías utilizar una **ArrayList** y una **LinkedList** en Java:

```

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Stack;
import java.util.Vector;

public class ListExample {
    public static void main(String[] args) {
        // ArrayList
        List<String> arrayList = new ArrayList<>();
        arrayList.add("Manzana");
        arrayList.add("Banana");

        // LinkedList
        LinkedList<String> linkedList = new LinkedList<>();
        linkedList.add("Cereza");
        linkedList.addFirst("Naranja");

        // Stack
        Stack<String> stack = new Stack<>();
        stack.push("Fútbol");
        stack.push("Baloncesto");

        // Vector
        Vector<String> vector = new Vector<>();
        vector.add("Gato");
        vector.add("Perro");

        // Imprimir el elemento superior de la pila (Stack)
        System.out.println("Top of Stack: " + stack.pop());

        // Imprimir el primer elemento de LinkedList
        System.out.println("First in LinkedList: " + linkedList.getFirst());
    }
}

```

Cada una de estas clases tiene su conjunto de características y puede ser más adecuada para diferentes situaciones dependiendo de los requisitos de rendimiento y de las operaciones que se necesiten realizar.

## 12.2. Set

Las colecciones **Set** representan un conjunto de elementos que no pueden tener duplicados. Es decir, cada elemento puede aparecer como máximo una sola vez en un **Set**. Este tipo de colecciones es útil cuando queremos asegurarnos de que no existen elementos repetidos y no nos importa el orden de los elementos.

Las principales características de las colecciones **Set** son:

- **Unicidad:** Los elementos que se añaden a un **Set** deben ser únicos. Si intentas añadir un elemento duplicado, este no se añade.
- **Desorden:** Los **Set** generalmente no garantizan el orden de los elementos. Por ejemplo, los elementos insertados no se almacenan en el orden en que se añadieron.
- **Operaciones básicas:** Incluyen añadir, eliminar y comprobar si un elemento está presente en el conjunto.

Las interfaces **Set** tienen varias implementaciones, pero las más comunes son **HashSet**, **LinkedHashSet** y **TreeSet**:

### 1. **HashSet**:

- Es la implementación más común de la interfaz **Set** y utiliza una tabla hash para almacenar los elementos.
- Es la opción más eficiente para operaciones de búsqueda, inserción y eliminación, con un tiempo constante en el caso promedio.
- No mantiene ningún orden para los elementos almacenados.

### 2. **LinkedHashSet**:

- Es una subclase de **HashSet** que además mantiene una lista doblemente enlazada a través de sus elementos.
- Mantiene el orden de inserción de los elementos, lo que significa que al iterar sobre un **LinkedHashSet**, los elementos se devolverán en el orden en que fueron insertados.
- Tiene un ligero aumento en el costo de rendimiento en comparación con **HashSet**.

### 3. **TreeSet**:

- Implementa la interfaz **NavigableSet** y almacena sus elementos en un árbol rojo-negro, que es una estructura de árbol balanceado.
- Los elementos se almacenan de manera ordenada según su orden natural o según un **Comparator** proporcionado en el momento de la creación del **TreeSet**.
- Las operaciones de inserción, eliminación y búsqueda tienen un tiempo de ejecución logarítmico.

### Ejemplo de uso de un Set en Java:

```
import java.util.HashSet;
import java.util.Set;

public class SetExample {
    public static void main(String[] args) {
        // Crear un conjunto de Strings
        Set<String> conjuntoDeFrutas = new HashSet<>();

        // Añadir elementos al conjunto
        conjuntoDeFrutas.add("Manzana");
        conjuntoDeFrutas.add("Banana");
        conjuntoDeFrutas.add("Cereza");
        conjuntoDeFrutas.add("Manzana"); // Este elemento no se añadirá, ya que es un duplicado

        // Imprimir los elementos del conjunto
        for (String fruta : conjuntoDeFrutas) {
            System.out.println(fruta);
        }
    }
}
```

En este ejemplo, se crea un **HashSet** de **String**, se añaden elementos a él, y se intenta añadir un elemento duplicado que no será incluido. Finalmente, se itera sobre el conjunto para imprimir sus elementos. Como es un **HashSet**, el orden de los elementos impresos puede no coincidir con el orden en que fueron añadidos. Las colecciones **Set** son muy útiles cuando necesitas una colección que no permita elementos repetidos y no requieras mantener un orden específico.

## 12.3. Queue

Las colecciones `Queue` en Java representan una secuencia de elementos que se procesan según el principio de "primero en entrar, primero en salir" (FIFO). Son útiles cuando necesitas mantener un orden específico para procesar los elementos, como en simulaciones de colas de espera, algoritmos de planificación de tareas o para implementar buffers.

Características principales de las colas (`Queue`):

- **Orden FIFO:** El primer elemento que se añade es el primero en ser retirado.
- **Operaciones de cola:** Proporcionan operaciones para añadir, eliminar y examinar elementos.
  - `offer(e)`: Añade un elemento a la cola de manera segura, retorna `false` si no puede añadir el elemento por limitaciones de capacidad.
  - `poll()`: Elimina y retorna el elemento de la cabeza de la cola o `null` si la cola está vacía.
  - `peek()`: Retorna el elemento de la cabeza de la cola sin eliminarlo, o `null` si la cola está vacía.

Las principales implementaciones de la interfaz `Queue` son:

- **LinkedList:** Además de implementar la interfaz `List`, `LinkedList` implementa `Queue` y puede usarse como una cola.
- **PriorityQueue:** Una cola de prioridad que ordena sus elementos según su orden natural o un comparador proporcionado en el momento de la creación.
- **ArrayDeque:** Una cola de doble extremo que permite añadir o eliminar elementos tanto al principio como al final de la cola.

Aquí tienes un ejemplo sencillo de cómo utilizar una cola en Java:

```
import java.util.Queue;
import java.util.LinkedList;

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> colaDeTareas = new LinkedList<>();

        // Añadir elementos a la cola
        colaDeTareas.offer("Lavar los platos");
        colaDeTareas.offer("Sacar la basura");
        colaDeTareas.offer("Limpiar las ventanas");

        // Procesar la cola
        while (!colaDeTareas.isEmpty()) {
            // Ver el próximo elemento a procesar sin eliminarlo
            String tarea = colaDeTareas.peek();
            System.out.println("Procesando tarea: " + tarea);

            // Eliminar y retornar el siguiente elemento en la cola
            colaDeTareas.poll();
        }
    }
}
```

En este ejemplo, se crea una `LinkedList` que actúa como una cola y se añaden tareas. Luego, se procesan y eliminan secuencialmente de la cola. Las colas son importantes cuando quieres asegurarte de que los elementos son procesados en el orden en que fueron agregados.

## 12.4. Map

Las colecciones **Map** en Java son estructuras de datos que almacenan pares de elementos en forma de claves y valores. Cada clave es única en el mapa y se utiliza para acceder al valor correspondiente. Las colecciones **Map** no son consideradas colecciones "verdaderas" en el sentido estricto, ya que no heredan de la interfaz **Collection**, pero forman parte del Java Collections Framework por su utilidad en la organización y manejo de datos asociativos.

Características principales de las colecciones **Map**:

- **Claves únicas:** No puedes tener claves duplicadas en un **Map**. Cada clave mapea exactamente a un solo valor.
- **Valores:** Los valores pueden estar duplicados. Diferentes claves pueden mapear al mismo valor.
- **Acceso y búsqueda:** Puedes acceder a los valores mediante sus claves, y buscar si ciertas claves o valores están presentes en el mapa.

Las principales implementaciones de la interfaz **Map** son **HashMap**, **LinkedHashMap**, **TreeMap** y **Hashtable**:

### 1. **HashMap**:

- Almacena los pares clave-valor en una tabla hash. Ofrece un rendimiento constante en tiempo para las operaciones básicas (get y put), asumiendo que la función hash dispersa adecuadamente los elementos.
- No garantiza ningún orden en la iteración.

### 2. **LinkedHashMap**:

- Es una extensión de **HashMap** que mantiene una lista doblemente enlazada a través de sus entradas.
- Mantiene el orden de inserción de los elementos o puede ser configurado para mantener el orden de acceso.
- Es ligeramente más lenta que **HashMap** para las operaciones básicas pero es muy eficiente en la iteración sobre sus elementos.

### 3. **TreeMap**:

- Implementa la interfaz **NavigableMap** y almacena sus elementos en un árbol rojo-negro.
- Ordena los elementos según su orden natural o según un **Comparator** proporcionado.
- Las operaciones de inserción, eliminación y búsqueda tienen un tiempo de ejecución logarítmico.

### Ejemplo de uso de un Map en Java:

```
import java.util.HashMap;
import java.util.Map;

public class MapExample {
    public static void main(String[] args) {
        // Crear un mapa de claves enteras a valores de cadena
        Map<Integer, String> codigoDeFrutas = new HashMap<>();

        // Añadir pares clave-valor al mapa
        codigoDeFrutas.put(1, "Manzana");
        codigoDeFrutas.put(2, "Banana");
        codigoDeFrutas.put(3, "Cereza");

        // Acceder a un valor utilizando su clave
        String fruta = codigoDeFrutas.get(1); // Devuelve "Manzana"

        // Imprimir todas las claves y los valores del mapa
        for (Map.Entry<Integer, String> entrada : codigoDeFrutas.entrySet()) {
            System.out.println("Clave: " + entrada.getKey() + ", Valor: " + entrada.getValue());
        }
    }
}
```

En este ejemplo, se crea un **HashMap** que asocia un código entero con un tipo de fruta (String). Luego, se añaden algunos pares clave-valor y se muestra cómo acceder a un valor mediante su clave. Finalmente, se itera sobre las entradas del mapa para imprimir todas las claves y valores. Las colecciones **Map** son especialmente útiles cuando necesitas buscar, actualizar o mantener una asociación entre pares de claves y valores.

## 13. Excepciones

Las excepciones en Java son eventos que alteran el flujo normal de ejecución de un programa. Son objetos que representan problemas que pueden ocurrir durante la ejecución de un programa, como errores de entrada/salida, intentos de acceso a índices de arrays fuera de límites, divisiones por cero, entre otros.

En Java, las excepciones se dividen en dos categorías principales:

1. **Excepciones comprobadas (checked exceptions):** Son aquellas que deben ser manejadas (capturadas) o declaradas en la firma del método mediante el uso de la cláusula `throws`. Estas excepciones son comprobadas en tiempo de compilación. Un ejemplo es `IOException`.
2. **Excepciones no comprobadas (unchecked exceptions):** Incluyen las excepciones que heredan de `RuntimeException`, y no necesitan ser declaradas ni capturadas obligatoriamente. Estas excepciones no son comprobadas en tiempo de compilación, por lo que el programador es responsable de evitarlas mediante una codificación cuidadosa. Un ejemplo es `NullPointerException`.

El manejo de excepciones en Java se realiza principalmente a través de cuatro palabras clave: `try`, `catch`, `finally`, y `throw`.

- **try:** Define un bloque de código en el que se pueden producir excepciones.
- **catch:** Define un bloque de código, conocido como manejador de excepciones, que se ejecutará cuando se produzca una excepción específica dentro del bloque `try`.
- **finally:** Define un bloque de código que se ejecutará después de los bloques `try` y `catch`, independientemente de si se produjo una excepción o no. Es útil para cerrar recursos.
- **throw:** Se utiliza para lanzar una excepción explícitamente.

Ejemplo de manejo de excepciones:

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int division = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Ocurrió un error aritmético: " + e.getMessage());
        } finally {
            System.out.println("Este bloque se ejecuta siempre.");
        }
    }
}
```

En este ejemplo, intentamos dividir un número entre cero, lo que causa una `ArithmeticException`. El bloque `catch` captura la excepción y ejecuta su código, que imprime un mensaje de error. Luego, el bloque `finally` se ejecuta y muestra otro mensaje. Este mecanismo permite que el programa responda de manera controlada a situaciones inesperadas sin terminar abruptamente.

## 13.1. Propagación, lanzamiento y creación de excepciones

### Propagación de excepciones

La propagación de excepciones se refiere a cómo una excepción no capturada en un método se transmite hacia arriba en la pila de llamadas de métodos. Si un método no maneja una excepción específica, la excepción es lanzada al método que lo invocó, y así sucesivamente, hasta que se encuentra un método que la maneje con un bloque `catch` o hasta que llega al método `main`. Si `main` tampoco la maneja, el programa terminará y se imprimirá una traza de la pila de llamadas.

### Lanzamiento de excepciones

El lanzamiento de excepciones ocurre cuando se usa la palabra clave `throw` para generar una excepción. Esto se puede hacer para señalar que se ha producido una situación excepcional que el método actual no está diseñado para manejar. Al lanzar la excepción, se crea una instancia de la misma y se "lanza" hacia arriba en la pila de llamadas para que otro método pueda manejarla.

Ejemplo de lanzamiento de excepción:

```
public void hacerAlgo(int valor) {
    if (valor < 0) {
        throw new IllegalArgumentException("El valor no puede ser negativo.");
    }
    // Resto del código...
}
```

### Creación de excepciones

La creación de excepciones se refiere al proceso de definir tus propias clases de excepción. Esto es útil cuando necesitas representar condiciones de error específicas de tu aplicación. Para crear una excepción personalizada, simplemente extiende la clase `Exception` o cualquier subclase de `Exception` (para excepciones comprobadas) o `RuntimeException` (para excepciones no comprobadas).

Ejemplo de creación de una excepción personalizada:

```
public class MiExcepcionPersonalizada extends Exception {
    public MiExcepcionPersonalizada(String mensaje) {
        super(mensaje);
    }
}
```

Luego, puedes usar `throw new MiExcepcionPersonalizada("mensaje")` para lanzar tu excepción personalizada donde sea necesario.

En resumen, la propagación es el proceso automático de pasar una excepción hacia arriba en la pila de llamadas, el lanzamiento es el acto de generar una excepción con `throw`, y la creación es la definición de nuevas clases de excepción para representar situaciones específicas.

## 13.2. Ejemplos comunes

Un resumen de algunas de las excepciones más comunes en Java, junto con situaciones típicas en las que podrían ocurrir:

1. **NullPointerException**: Se lanza cuando se intenta utilizar una referencia que apunta a null en un caso que requiere un objeto real, como llamar a un método o acceder a un campo de un objeto nulo.
2. **ArrayIndexOutOfBoundsException**: Se lanza cuando se intenta acceder a un índice de un array que está fuera de los límites permitidos (es decir, menor que cero o mayor o igual que el tamaño del array).
3. **StringIndexOutOfBoundsException**: Similar a **ArrayIndexOutOfBoundsException**, pero específica para los índices de cadenas de caracteres, como cuando se intenta acceder a un índice de una cadena que no existe.
4. **ArithmeticException**: Se lanza, por ejemplo, cuando se realiza una división por cero en operaciones aritméticas enteras.
5. **IllegalArgumentException**: Se lanza para indicar que se ha pasado un argumento ilegal o inapropiado a un método.
6. **ClassCastException**: Se lanza al intentar realizar un cast de un objeto a una clase de la cual no es una instancia, en otras palabras, cuando la conversión de tipos es inválida.
7. **IOException**: Se lanza para manejar errores de entrada/salida, como problemas al leer o escribir en archivos, o al manejar conexiones de red.
8. **FileNotFoundException**: Es una subclase de **IOException** y se lanza cuando se intenta acceder a un archivo o directorio que no existe.
9. **NumberFormatException**: Se lanza al intentar convertir una cadena a un tipo numérico, pero la cadena no tiene el formato adecuado.
10. **InterruptedException**: Se lanza cuando un hilo que está esperando, durmiendo o de otra manera ocupado, es interrumpido por otro hilo.

Estos son solo algunos ejemplos de las excepciones más comunes que te encontrarás mientras programas en Java. Cada una de estas excepciones hereda de la clase **Throwable** y son parte de la jerarquía de excepciones que Java utiliza para clasificar y manejar errores y otras condiciones excepcionales.

## 14. Streams

En informática, el término "flujos de datos" (o "streams" en inglés) se refiere a secuencias de datos procesadas de manera continua y secuencial. En el contexto de Java, los flujos de datos son utilizados para leer o escribir datos de forma eficiente, ya sea desde o hacia fuentes externas como archivos, redes o entre programas.

Java proporciona varias clases en el paquete `java.io` para manejar flujos de datos, que se clasifican principalmente en dos tipos:

1. **Flujos de Entrada (Input Streams):** Se utilizan para leer datos de una fuente. Ejemplos incluyen `FileInputStream`, para leer datos de archivos, y `BufferedReader`, para leer texto de una secuencia de entrada de manera eficiente.
2. **Flujos de Salida (Output Streams):** Se utilizan para escribir datos a un destino. Ejemplos incluyen `FileOutputStream`, para escribir datos en archivos, y `BufferedWriter`, para escribir texto a una secuencia de salida de forma eficiente.

El manejo de flujos de datos es fundamental para muchas aplicaciones que requieren la entrada y salida de información, y su uso adecuado permite gestionar recursos como archivos y conexiones de red de manera segura y efectiva.

## 14.1. Flujos de caracteres

En Java, los flujos de caracteres se refieren a las clases y métodos que se utilizan para manejar la entrada y salida de texto. A diferencia de los flujos de bytes que leen o escriben datos binarios, los flujos de caracteres están diseñados para procesar caracteres Unicode y se adaptan automáticamente a las codificaciones de caracteres locales.

Los flujos de caracteres en Java se dividen en dos categorías:

1. **Readers:** Son clases diseñadas para leer caracteres. `Reader` es la clase abstracta principal para la lectura de flujos de caracteres. Ejemplos de implementaciones concretas incluyen `FileReader` para leer archivos de texto y `BufferedReader`, que agrega funcionalidad de buffering para mejorar la eficiencia en la lectura.
2. **Writers:** Son clases diseñadas para escribir caracteres. `Writer` es la clase abstracta principal para la escritura de flujos de caracteres. Ejemplos de implementaciones concretas incluyen `FileWriter` para escribir en archivos de texto y `BufferedWriter`, que proporciona un buffer para mejorar el rendimiento en la escritura.

Estos flujos son esenciales cuando se trabaja con texto, ya que aseguran que los caracteres se manejen correctamente de acuerdo con su codificación, lo cual es crucial para la internacionalización y localización de aplicaciones.

## 14.2. Flujos de bytes

En Java, los flujos de bytes se utilizan para realizar operaciones de entrada y salida de datos binarios. Se trata de flujos que manejan datos primitivos de tipo byte, lo cual es útil cuando se trabaja con archivos binarios, transmisiones de red y cualquier otra fuente o destino que requiera manejar los datos en su forma más cruda y sin procesar.

Los flujos de bytes se dividen principalmente en dos categorías:

1. **InputStreams:** Son utilizados para leer datos binarios de una fuente. La clase base para todos los **InputStreams** es **InputStream**. Ejemplos de clases concretas son **FileInputStream** para leer datos de un archivo y **BufferedInputStream** que añade una capa de buffering para mejorar la eficiencia en la lectura.
2. **OutputStreams:** Son utilizados para escribir datos binarios hacia un destino. La clase base para todos los **OutputStreams** es **OutputStream**. Ejemplos de clases concretas son **FileOutputStream** para escribir datos en un archivo y **BufferedOutputStream** que proporciona un buffer para mejorar el rendimiento en la escritura.

Estos flujos de bytes son fundamentales para la manipulación de datos a bajo nivel, y son ampliamente utilizados para todo tipo de operaciones de E/S que no involucran caracteres de texto.

## 14.3. Tabla Resumen

### Resumen de clases para ficheros de texto y binarios

	Texto	Binario
Archivo	<code>File archivo = new File("C:\\fichero.txt");</code>	<code>File archivo = new File("C:\\fichero.bin");</code>
Lector	<code>FileReader lector = new FileReader(archivo);</code>	<code>FileInputStream flujoEntrada = new FileInputStream(archivo);</code>
Lector con Buffer	<code>BufferedReader lectorConBuffer = new BufferedReader(lector);</code>	<code>BufferedInputStream lectorConBuffer = new BufferedInputStream(flujoEntrada);</code>
Archivo	<code>String archivo = "C:\\fichero.txt";</code>	<code>String archivo = "c:\\fichero.bin";</code>
Escritor	<code>FileWriter escritor = new FileWriter(archivo);</code>	<code>FileOutputStream flujoSalida = new FileOutputStream(archivo);</code>
Escritor con Buffer	<code>BufferedWriter escritorConBuffer = new BufferedWriter(escritor);</code>	<code>BufferedOutputStream escritorConBuffer = new BufferedOutputStream(flujoEntrada);</code>

## 15. Ficheros

Los tipos de ficheros pueden determinar cómo se leen, se escriben y se procesan los datos. Aquí tienes los tipos de ficheros más comunes:

1. **Ficheros de Texto:** Contienen datos que están estructurados como texto plano y pueden leerse con un editor de texto. Los caracteres en estos ficheros suelen estar codificados en formatos como ASCII o UTF-8. Ejemplos de extensiones son `.txt`, `.csv` o `.html`.
2. **Ficheros Binarios:** Contienen datos en un formato que no es texto plano y generalmente requieren un programa específico para ser interpretados. Pueden almacenar imágenes, audio, video, ejecutables, o cualquier otro tipo de datos en una forma binaria. Ejemplos de extensiones son `.exe`, `.png`, `.mp3`, entre otros.

## 15.1. Clase File

La clase `File` en Java es parte del paquete `java.io` y representa la abstracción de los nombres de archivos y directorios de manera independiente del sistema operativo. Esta clase no se utiliza para leer o escribir datos en archivos; su propósito principal es manejar los metadatos del archivo o directorio, como nombres, rutas y permisos.

Aquí tienes un resumen de algunos de los métodos más comunes de la clase `File`:

- `exists()`: Verifica si el archivo o directorio representado por este objeto `File` existe.
- `createNewFile()`: Crea un archivo nuevo vacío con el nombre del objeto `File` si no existe.
- `mkdir()`: Crea un directorio nombrado por este objeto `File`.
- `mkdirs()`: Crea el directorio nombrado por este objeto `File`, incluyendo cualquier directorio padre necesario.
- `list()`: Devuelve un array de strings con los nombres de los archivos y directorios en el directorio representado por este objeto `File`.
- `isFile()`: Verifica si el objeto `File` representa un archivo en el sistema de archivos.
- `isDirectory()`: Verifica si el objeto `File` representa un directorio en el sistema de archivos.
- `delete()`: Elimina el archivo o directorio representado por este objeto `File`.
- `getPath()`: Devuelve la ruta del archivo o directorio como una cadena de texto.
- `getAbsolutePath()`: Devuelve la ruta absoluta del archivo o directorio.
- `canRead()`: Verifica si la aplicación puede leer el archivo o directorio representado por este objeto `File`.
- `canWrite()`: Verifica si la aplicación puede escribir en el archivo o directorio representado por este objeto `File`.
- `length()`: Devuelve el tamaño del archivo en bytes.
- `renameTo(File dest)`: Renombra el archivo representado por este objeto `File` al nombre representado por el objeto `File` de destino.

Esta es una visión general, y hay muchos más métodos disponibles en la clase `File` para realizar distintas operaciones relacionadas con el sistema de archivos. Es importante recordar manejar las posibles excepciones que estos métodos pueden lanzar, como `IOException`, cuando se trabaja con archivos.

## 15.2. Acceso secuencial

El acceso secuencial a ficheros es un método de lectura y escritura de datos en un archivo donde las operaciones se realizan en un orden secuencial, comenzando desde el principio del archivo y avanzando hacia el final. Este tipo de acceso es simple y eficiente cuando necesitas procesar un archivo completo de principio a fin.

En un acceso secuencial, no puedes saltar directamente a una posición específica del archivo sin antes pasar por todos los datos anteriores. Esto lo diferencia del acceso aleatorio, donde puedes moverte directamente a cualquier parte del archivo sin leer las partes intermedias.

En Java, puedes realizar acceso secuencial a ficheros utilizando clases del paquete `java.io` como `FileInputStream` y `FileOutputStream` para archivos binarios, o `FileReader` y `FileWriter` para archivos de texto. Estas clases permiten leer o escribir datos en el archivo de manera continua.

El acceso secuencial es particularmente útil para archivos grandes donde no es práctico o necesario cargar todo el contenido en memoria antes de procesarlo. Ejemplos típicos de uso incluyen procesar logs de eventos, operaciones en archivos de registro, o la lectura de datos en formatos estructurados como CSV o JSON que se procesan línea por línea.

### Ejemplo de Escritura a un Archivo

Aquí tienes un ejemplo simple de cómo escribir en un archivo de texto usando `FileWriter`:

```
import java.io.FileWriter;
import java.io.IOException;

public class EscrituraArchivo {
    public static void main(String[] args) {
        String rutaArchivo = "ejemplo.txt";
        try (FileWriter writer = new FileWriter(rutaArchivo)) {
            writer.write("Hola, esto es un texto de ejemplo.\n");
            writer.write("Aquí añadimos otra línea de texto.");
        } catch (IOException e) {
            System.out.println("Ocurrió un error al escribir en el archivo: " + e.getMessage());
        }
    }
}
```

Este código crea un archivo llamado `ejemplo.txt` y escribe dos líneas de texto en él. El uso de `try-with-resources` asegura que el `FileWriter` se cierre automáticamente al final del bloque `try`, incluso si ocurre una excepción.

### Ejemplo de Lectura de un Archivo

Para leer el contenido de un archivo, puedes usar `FileReader` junto con `BufferedReader`, que proporciona una forma eficiente de leer líneas de texto:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class LecturaArchivo {
    public static void main(String[] args) {
        String rutaArchivo = "ejemplo.txt";
        try (BufferedReader reader = new BufferedReader(new FileReader(rutaArchivo))) {
            String linea;
            while ((linea = reader.readLine()) != null) {
                System.out.println(linea);
            }
        } catch (IOException e) {
            System.out.println("Ocurrió un error al leer el archivo: " + e.getMessage());
        }
    }
}
```

Este código lee todas las líneas del archivo `ejemplo.txt` y las imprime en la consola. Al igual que en el ejemplo de escritura, se utiliza `try-with-resources` para asegurar que el `BufferedReader` se cierre correctamente después de su uso.

## OpenOption

En Java, `OpenOption` es una interfaz que representa las opciones de configuración que se pueden usar al abrir o crear un archivo con clases que implementan esta interfaz, como `Files` y sus métodos asociados. Proporciona un mecanismo para especificar cómo debe comportarse la

operación de apertura en diferentes situaciones.

Algunas de las implementaciones más comunes de `OpenOption` son:

- **StandardOpenOption**: Define varias opciones estándar que se pueden usar al abrir un archivo. Algunas de estas opciones incluyen:
  - **READ**: Abre el archivo para lectura.
  - **WRITE**: Abre el archivo para escritura. Si no existe el archivo, se crea.
  - **APPEND**: Si se escribe en el archivo, los datos se escriben al final.
  - **CREATE**: Crea un nuevo archivo si no existe.
  - **CREATE\_NEW**: Crea un nuevo archivo, fallando si ya existe.
  - **DELETE\_ON\_CLOSE**: El archivo se elimina automáticamente cuando se cierra el stream.
  - **TRUNCATE\_EXISTING**: Si el archivo ya existe y se abre para escribir, reduce su tamaño a 0.

Cuando usas métodos como `Files.newInputStream(Path, OpenOption...)` o `Files.newOutputStream(Path, OpenOption...)`, puedes pasar cero o más `OpenOption` como argumentos para definir el comportamiento del archivo que estás abriendo o creando. Por ejemplo:

```
Path path = Paths.get("miArchivo.txt");

// Usando StandardOpenOption para crear un nuevo archivo y escribir en él
try (OutputStream out = Files.newOutputStream(path, StandardOpenOption.CREATE, StandardOpenOption.WRITE)) {
    // Escribir datos en el archivo
}
```

Estas opciones proporcionan un control detallado sobre el comportamiento de los archivos en Java, permitiendo a los desarrolladores adaptar la operación de archivos a sus necesidades específicas.

## 15.3. Acceso aleatorio

El acceso aleatorio a ficheros en Java permite leer y escribir datos en cualquier posición de un archivo sin necesidad de recorrerlo secuencialmente desde el principio. Este tipo de acceso es muy útil cuando necesitas manipular grandes volúmenes de datos o cuando necesitas actualizar partes específicas de un archivo sin reescribir todo el contenido.

Para implementar el acceso aleatorio en Java, puedes utilizar la clase `RandomAccessFile` del paquete `java.io`. Esta clase soporta tanto la lectura como la escritura en un archivo, y permite posicionarte en cualquier punto del archivo usando un índice de byte.

Con `RandomAccessFile`, puedes:

- Abrir un archivo en modo de lectura ("`r`"), escritura ("`w`") o lectura/escritura ("`rw`").
- Mover el puntero de archivo a una posición específica con el método `seek(long pos)`.
- Leer datos del archivo con métodos como `readInt()`, `readByte()`, `readUTF()`, entre otros.
- Escribir datos en el archivo usando métodos como `writeInt()`, `writeByte()`, `writeUTF()`, etc.

El acceso aleatorio es especialmente poderoso en aplicaciones como bases de datos donde los registros necesitan ser accedidos y modificados frecuentemente y de manera no secuencial.

### Ejemplo de Escritura y Lectura con RandomAccessFile

Vamos a escribir algunos datos en un archivo y luego leeremos esos mismos datos especificando su posición exacta en el archivo.

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class AccesoAleatorio {
    public static void main(String[] args) {
        String rutaArchivo = "datos.dat";

        try (RandomAccessFile raf = new RandomAccessFile(rutaArchivo, "rw")) {
            // Escribir algunos datos en el archivo
            raf.writeInt(123); // 4 bytes para un int
            raf.writeDouble(45.67); // 8 bytes para un double
            raf.writeUTF("Hola Mundo"); // UTF usa 2 bytes + longitud del texto

            // Leer los datos
            // Volver al comienzo del archivo
            raf.seek(0);

            // Leer los datos en el mismo orden que fueron escritos
            int numero = raf.readInt();
            double flotante = raf.readDouble();
            String texto = raf.readUTF();

            System.out.println("Número entero: " + numero);
            System.out.println("Número flotante: " + flotante);
            System.out.println("Texto: " + texto);

            // Ejemplo de acceso aleatorio:
            // Ir directamente a la posición del double y leerlo
            raf.seek(4); // int tiene 4 bytes, por lo que saltamos los primeros 4 bytes
            double flotanteDirecto = raf.readDouble();
            System.out.println("Número flotante leído directamente: " + flotanteDirecto);

        } catch (IOException e) {
            System.out.println("Ocurrió un error al acceder al archivo: " + e.getMessage());
        }
    }
}
```

En este ejemplo, el archivo `datos.dat` se abre en modo lectura/escritura ("`rw`"). Primero, escribimos un entero, un doble y una cadena. Luego, usamos el método `seek` para posicionar el puntero al inicio del archivo y leer los datos en el orden en que fueron escritos. Además, demostramos cómo acceder directamente a una posición específica del archivo para leer un valor, en este caso, un `double` que estaba después de un `int`.

## 15.4. Serialización

La serialización en Java es el proceso de convertir un objeto en una secuencia de bytes que puede ser guardada en un archivo o transmitida a través de la red. Este proceso también permite que un objeto pueda ser posteriormente reconstruido (deserializado) a su estado original, leyendo la secuencia de bytes y creando un objeto en memoria.

Java proporciona un mecanismo de serialización estándar que es automático para los objetos que implementan la interfaz `Serializable`. Cuando una clase implementa esta interfaz, está marcando que sus objetos pueden ser serializados. No obstante, hay que tener en cuenta que todos los atributos de la clase que no son `transient` ni `static` serán parte del proceso de serialización.

Aquí tienes un ejemplo corto de cómo serializar y deserializar un objeto:

```
import java.io.*;

class Persona implements Serializable {
    private static final long serialVersionUID = 1L; // Esto asegura una deserialización compatible
    private String nombre;
    private transient int edad; // 'transient' significa que este campo no será serializado

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String toString() {
        return "Persona{nombre='" + nombre + "', edad=" + edad + "}";
    }
}

public class SerializacionEjemplo {
    public static void main(String[] args) {
        Persona persona = new Persona("Juan", 30);

        // Serializar objeto
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("persona.ser"))) {
            oos.writeObject(persona);
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserializar objeto
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("persona.ser"))) {
            Persona personaLeida = (Persona) ois.readObject();
            System.out.println(personaLeida);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

En este ejemplo, la clase `Persona` implementa `Serializable`, lo que permite que sus instancias sean serializadas y deserializadas. El campo `edad` no se serializará debido al modificador `transient`. El proceso de serialización se lleva a cabo a través de `ObjectOutputStream` y la deserialización a través de `ObjectInputStream`.

Es importante utilizar la serialización con cuidado, ya que puede tener implicaciones de seguridad y es fundamental garantizar la compatibilidad de versiones de las clases serializadas. Por eso, se recomienda definir un `serialVersionUID` estático para cada clase serializable.